Network Coding Parallelization Based on Matrix Operations for Multicore Architectures

Simon Wunderlich and Juan Cabrera and Frank H.P. Fitzek
Chair of Communication Networks
TU Dresden
simon.wunderlich@mailbox.tu-dresden.de
[juan.cabrera|frank.fitzek]@tu-dresden.de

Morten V. Pedersen
Dept. of Electronic Systems
Aalborg University
mvp@es.aau.dk

***Abstract*—Network coding has the potential to improve the performance of current and future communication systems (including transportation and storage) and is currently even considered for communication architectures between the individual processors on same board or different boards in close proximity. Despite the fact that single core implementation show already comparable coding speeds with standard coding approaches, this paper is push network coding to the next level by exploiting multicore architectures. The disruptive idea presented in the paper is to break with current software implementation and coding approaches and to adopt highly optimized dense matrix operations from the high performance computation field for network coding in order to increase the coding speed. The paper presents the novel coding approach for multicore architectures and shows coding speed gains on a commercial platform such as the Raspberry Pi2 with four cores in the order of up to one full magnitude. The speed increase even outperforms the number of cores of the Raspberry Pi2 with four cores as the newly introduced approach is exploiting the cache architecture way better than by the book matrix operations.**

I. Introduction

Random Linear Network Coding (RLNC) [1] is a popular approach to reliably and efficiently transfer data in complex, chaotic or lossy networks such as wireless mesh networks [2] or satellite links [3]. Other applications of RLNC include data storage where blocks of data can be scattered over arbitrary types of storage servers. The main idea of RLNC is to divide data into separate symbols and combine these symbols linearly based on randomized coefficients. A receiver only needs to get a sufficiently large (linearly independent) amount of these coded symbols to be able to solve the linear system and *decode* the original data.

RLNC allows to minimize the required feedback from the receiver to the sender and shows good performance for many topologies and loss patterns. One major drawback is the computational complexity for encoding and decoding, which may limit the maximum throughput on the devices. On the other hand, the current trend in hardware is going towards adding more cores not only in PCs and Laptops but also in embedded hardware like smart phones, TVs or WiFi access points to maximize performance while decreasing power consumption by temporarily disabling idle cores. It is therefore interesting to exploit recent CPU features such as SIMD instructions and multicore architectures where multiple, possibly different CPUs work together to perform encoding or decoding.

In the HAEC research project [4] at TU Dresden, network coding is even considered to operate on multiple cores on the same and on different boards. The current implementation for network coding, namely KODO [5] by Steinwurf, is optimizing for low complexity on a single core. Therefore this paper is disruptive as it introduces a new way to perform the network coding.

The objective of this work is to evaluate multicore optimizations which are common in High Performance Computing (HPC) for RLNC to allow high throughput even on embedded devices, without the typical requirement of high synchronicity of tasks to minimize *idle waiting* for other processes and allow seamless integration into operating systems which are not exclusively used for network processing.

The paper is structured as followed: Network Coding basics as used in our application is reviewed in Section II. Optimized matrix operations and our application on RLNC is described in Section III. Experimental results and micro-benchmark results are presented in Section IV. The conclusion and future work is presented in Section V.

II. Network Coding basics

In RLNC, data is coded by the sender and decoded by the receiver by performing linear operations in Galois fields. Typically, the original data is divided into a number of packets n of length m. The packets are further grouped into generations, whereas one generation contains g packets. The original data can be described as a matrix M of with g rows and m columns, where each row represents one packet of length m.

To create an encoded data packet, one needs to generate a vector c with g random coefficients and multiply it with the data matrix M to obtain a coded symbol x. Similarly, to create multiple coded packets, a matrix with random entries, $g + r$ rows and g columns can be created and multiplied with the data matrix M to obtain $g + r$ coded symbols stored in a matrix X. r represents the number of redundant packets to be generated, since at least g coded packets have to be received to reconstruct the original data by the receiver. Encoding can therefore be expressed as matrix multiplication:

$$X = C \cdot M$$

Each coded packed is separately sent with its corresponding coding vector c and the coded symbol x, represented by the rows of the respective matrices. Once the receiver got at least g coded packets with linearly independent coding vectors, it can reconstruct the original data matrix M. By placing the coding vectors into a matrix $\bar{C}$ and into the coded symbols $\bar{X}$, the original symbols M can be reconstructed by calculating:

$$M = \bar{C}^{-1} \cdot \bar{X}$$

Note that the encoding and decoding operation share the matrix multiplication step. For the decoding operation, the encoding vector matrix additionally has to be inverted. Many applications implement decoding by combining the inversion and matrix multiplication, e.g. by performing the Gauss-Jordan algorithm on the C at the same time while applying row operations on X. These row operations on X can be parallelized for GPUs [6] or SMP systems [7], however speed up is limited for smaller block sizes (≤ 2048 byte) since working with multiple threads on the same coded symbol requires tight synchronization. In this paper, we explicitly invert the matrix C first to re-use the optimized matrix multiplications and implement another optimized inversion separately to overcome this limitation.

III. Optimized matrix operations reviewed

Matrix multiplication and inversion are standard operations in many scientific applications and are therefore very well researched. Standard interfaces like BLAS [8], [9] and LAPACK [10] exist for decades and provide many common vector and matrix operation. Optimized and self-optimizing libraries such as Goto [11] or ATLAS [12] take cache hierarchies, TLB and SIMD instructions into account to maximize performance. However these libraries operate on floating point numbers, not on Galois fields. Libraries such as FFLAS/FFPACK [13] or Linbox [14] exist to leverage these highly optimized BLAS implementations for various finite field variants by converting elements to floating point numbers and back. This approach yields very good results for big problem sizes, but the conversions adds considerable overhead for small problem sizes. Also "small" Galois field sizes such as GF(2) or GF(2^8) can perform better in native implementations by leveraging SIMD to perform much more operations simultaneously in one instruction. The approach presented in this paper therefore adopts ideas and conventions of efficient BLAS operations, but re-implements them for Galois fields.

One typical optimization is to operate on square sub-blocks of matrices rather than full rows and lines of a matrix when doing e.g. matrix multiplication. Working on square blocks improves the spatial locality of the data and maximizes the operations per fetched data, at least for $O(n^3)$ algorithms [15]. The optimal block size depends on the architecture and is usually chosen so that all operands fit in the L1 cache and should

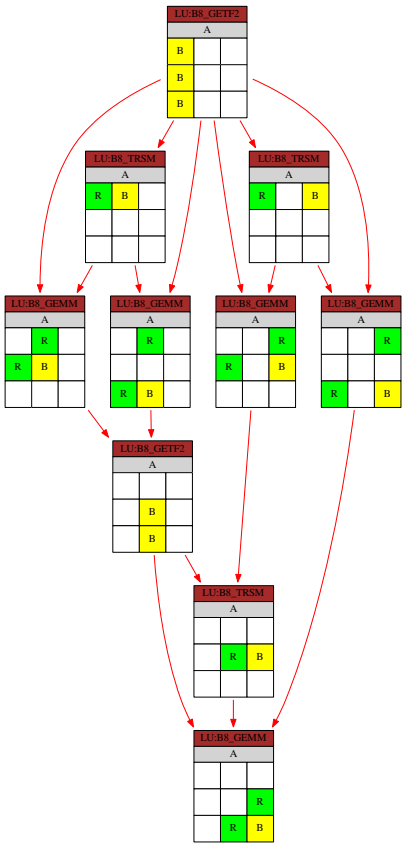

Fig. 1. Matrix inversion stage 1: LU factorization of the input matrix

be a multiple of the architectures SIMD operation size. Further possible optimizations include multiple levels of blocking to match L2 and L3 caches or the TLB, recursive blocking and reordering the input data to adapt to the algorithms access pattern [16]. However, we focused on a single level of blocking for this work to keep the algorithm simple.

As for the matrix inversion, a blocked version based on LU factorization has been implemented, similar to the LAPACK GETRI routine. The three main stages are:

1) LU factorization of the input matrix
2) Inversion of the upper matrix U
3) Solving $A^-1 \cdot L = U^-1$ for A^-1

Each step involves various operations which operate on sub-blocks of the matrix, such as matrix-matrix multiplication, matrix-triangle matrix multiplication, triangle-matrix system solving, etc. These few base matrix operations "kernels" can be individually optimized e.g. using SIMD operations.

Each of these block matrix operations can be considered a separate task with *inputs* and *outputs* in the memory. Jack Dongarra et al. describe how to exploit these data dependencies among tasks to parallelize the matrix inversion very efficiently [17]. The basic idea is to first formulate the algorithm conventionally. Data operations on the subblocks are then recorded, and the resolved data dependencies are

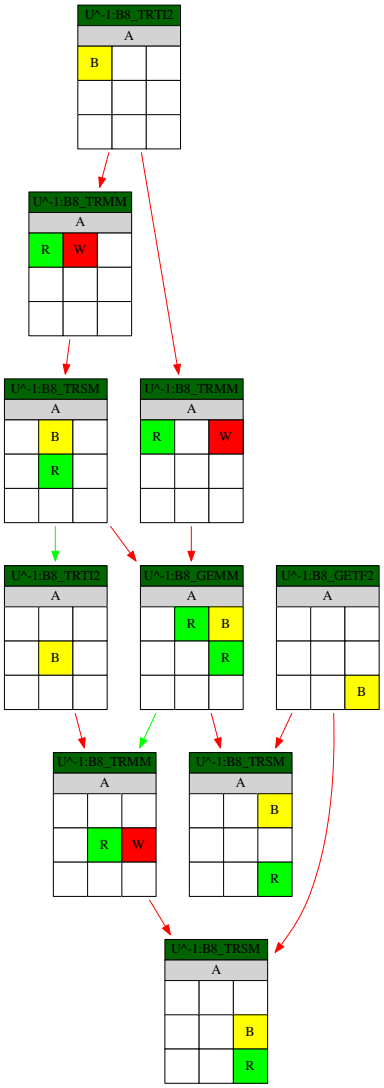

Fig. 2. Matrix inversion stage 2: Inversion of the upper matrix U

formulated in a directed acyclic graph (DAG) with nodes representing the individual tasks and edges representing the data dependencies among the nodes. To actually perform the matrix inversion, a scheduler distributes tasks with satisfied dependencies until the DAG is completely processed.

This method has quite a few interesting properties: The synchronization solely depends on data dependencies, no *artificial* synchronization points need to be inserted, not even between the different stages. The task creation overhead is minimized as the scheduler creates threads based on the number of used cores on startup, and each thread just accesses the task queue to obtain a new task to work on, which is substantially faster than creating new threads. Finally, this approach can cope very well with cores on different speeds (e.g. Big Little architectures) and systems where some cores may be busy with other tasks, e.g. IO or processing the received data.

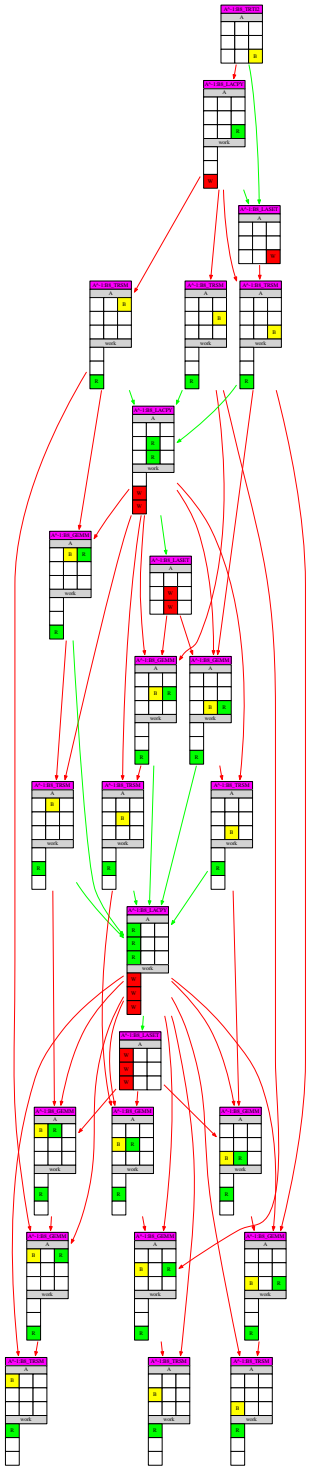

Fig. 3. Matrix Inversion stage 3: Solving $A^-1 \cdot L = U^-1$ for A^-1

The DAG created for the matrix inversion is shown in Figure 1, 2 and 3 respectively for a matrix inversion on 3x3 blocks. For each node, the involved blocks have been colored: green blocks with letter 'R' are blocks where data is read from, red blocks with letter 'W' are written to, and on yellow

blocks with letter 'B', both operations are performed. Note that pivoting has not been implemented yet. The DAG for the matrix multiplication step is created in the same way by recording the various matrix-matrix multiplications of the respective sub-blocks. Since all sub-blocks of the output matrix are computed independently in this step, the corresponding DAG provides much more parallelism to exploit.

IV. Results

Experimental results have been performed on the Raspberry Pi 2 Model B board, which features four ARM Cortex-A7 cores in a Broadcom BCM2836 System on Chip (SoC) system, clocked at 900 MHz. Each core features 32 KiB L1 Data cache and 32 KiB L1 Instruction cache, and all cores share 512 KiB Level 2 cache. Furthermore, these cores support the NEON extension, which provides an 128-bit SIMD instruction set which can greatly speed up the GF(2^8) operations as described by H.P. Anvin [18] or Plank [19] in greater detail. All benchmark results (including baseline results) have been performed with NEON-enabled code which have been adopted from the fifi/kodo library [5].

The non-blocked baseline test contains a non-blocked SIMD matrix multiplication *by the book* which represents the encoding of one generation using one thread and without call recording. The blocked baseline test contains the same matrix multiplication using the blocked method as described in Section III to work more cache-efficient, also using one single thread and without call recording.

For the encoding tests the blocked method for matrix multiplication is used while performing task recording and scheduling on multiple threads. For the decoding tests, the same matrix multiplication and additionally the inversion of the coding vector matrix is performed. Note that time includes recording, dependency resolving and scheduling of tasks. In practice, it would be sufficient to perform the recording and dependency resolving only once and copy the resulting DAG for each generation.

Figure 4, 5 and 6 show the throughput performance in MiB per second for the described test cases for generation sizes / matrix sizes g = 1024, 128 and 16 respectively. The throughput is calculated as the ratio of sending g packets of size 1536 byte divided by the time to complete the encoding or decoding process. All measurements have been performed with different block sizes ($nb = 16, 32, 64, ...1024$ as applicable), and only the best results have been plotted. These were either block sizes of 16 or 32, since these allow to keep the operands in the L1 data cache.

For g = 16, the blocked variant shows almost no gain compared to the non-blocked variant, since the data fits in the L1 cache of one core. The scheduled variants show degraded throughput for one thread, since the recording and dependency resolving create additional overhead. Two threads show a little gain compared to the baseline measurements, four threads don't add much more since the problem size is too small.

For g = 128, blocking already gains 15% compare to the non-blocked baselines due to cache effects. The recording and scheduling overhead is insignificant as the encoding with one thread compared to the blocked baseline measurement shows. Using four threads results in 2.82 times speed up over the single thread encoding, and 2.4 times speed up over for decoding. Note that decoding is naturally slower since the inversion of the matrix is an additional task to be done, but the difference to encoding, performing with only 71% throughput compared to encoding for four threads.

For g = 1024, the blocked variant greatly outperforms the non-blocked variant by factor 2.83. Using four threads for encoding is 3.7 times faster for encoding and 3.05 times faster for decoding compared to the one thread version. Encoding is more than ten times faster than the non-blocked baseline encoding. Since the matrix inversion step is now considerably more complex compared to the multiplication step, the decoding is also slower and performs at only 46% throughput compared to encoding for four threads. As expected, recording and scheduling costs are also negligible for this generation size.

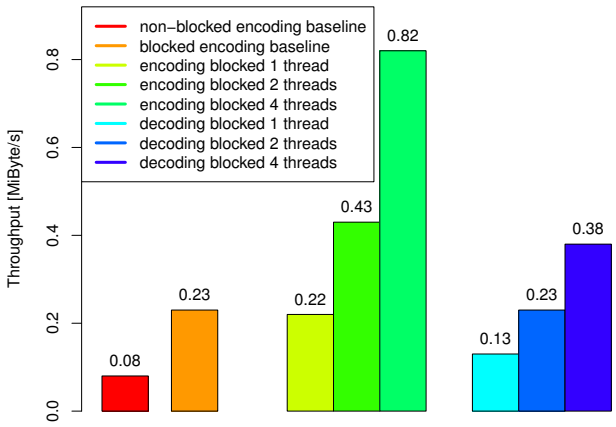

Fig. 4. Encoding and Decoding performance for g = 1024

Also note that the matrix multiplication step with the evaluated block size of 1536 takes is the more expensive operation compared to the matrix inversion, as Table I illustrates, and therefore benefits most from optimizations.

g	multiplication (ms)	inversion (ms)	ratio
16	1.703	0.345	4.9
32	6.573	0.914	7.2
64	21.341	3.479	6.1
128	82.326	17.411	4.7
256	336.398	106.861	3.1
512	1548.750	659.469	2.3
1024	6730.380	5166.920	1.3

TABLE I
MULTIPLICATION AND INVERSION RUN-TIMES FOR DIFFERENT GENERATION SIZES WITH 1 THREAD

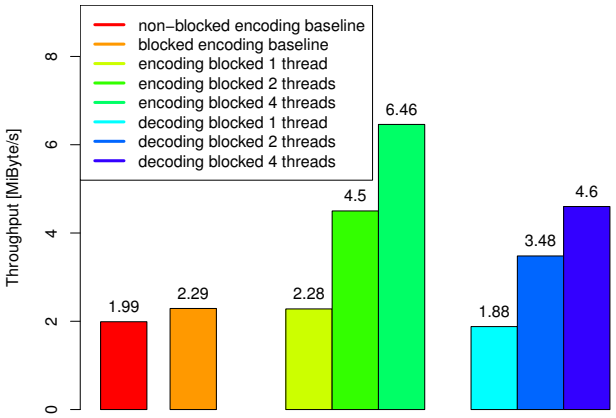

Fig. 5. Encoding and Decoding performance for g = 128

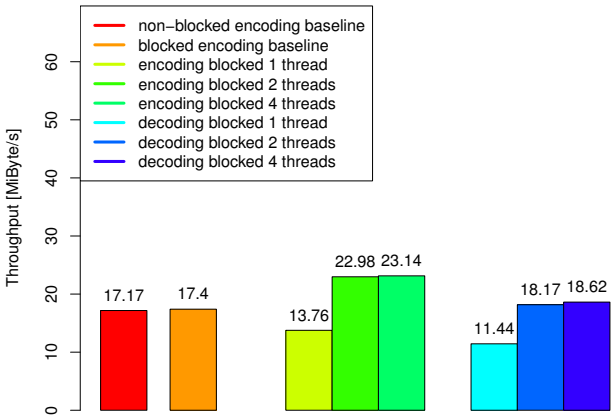

Fig. 6. Encoding and Decoding performance for g = 16

V. Conclusion

In this paper we have shown how optimization approaches for HPC dense matrix applications can be applied for RLNC. The implemented micro benchmarks showed very good performance and core utilization with speed ups up to factor 10 by using multiple cores and cache optimizations. The proposed approach of recording and scheduling fine grained tasks is not limited to the LU factorization based matrix inversion and multiplication, but can also be applied for any other fixed computation task. Furthermore, the minimized synchronization requirements makes the implementation ideal for real world applications where processors perform other tasks next to coding.

Our future work will focus on systems with even more and different kind of cores like the BIG.little ARM architectures. The scheduler can be improved further to consider characteristics of the graph, like preferring important tasks on the *hot path* or considering data locality. Our final goal is the integration into the HAEC [4] system.

VI. Acknowledgments

This work is supported in a part by the German Research Foundation (DFG) in the Collaborative Research Center 912 ***H**ighly **A**daptive **E**nergy–Efficient **C**omputing (HAEC).*

References

[1] T. Ho, R. Koetter, M. Medard, D. R. Karger, and M. Effros, "The benefits of coding over routing in a randomized setting," 2003.

[2] P. Pahlevani, M. Hundebøll, M. V. Pedersen, D. E. Lucani, H. Charaf, F. H. Fitzek, H. Bagheri, and M. Katz, "Novel concepts for device-to-device communication using network coding," *Communications Magazine, IEEE*, vol. 52, no. 4, pp. 32–39, 2014.

[3] U. Speidel, P. Vingelmann, J. Heide, M. Médard *et al.*, "Can network coding bridge the digital divide in the pacific?" *arXiv preprint arXiv:1506.01048*, 2015.

[4] S. 912, "Highly adaptive energy–efficient computing," http://tu-dresden.de/forschung/forschungskompetenz/sonderforschungsbereiche/sfb912.

[5] M. V. Pedersen, J. Heide, and F. H. Fitzek, "Kodo: An open and research oriented network coding library," in *Networking 2011 Workshops*. Springer, 2011, pp. 145–152.

[6] H. Shojania, B. Li, and X. Wang, "Nuclei: Gpu-accelerated many-core network coding," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 459–467.

[7] H. Shojania and B. Li, "Parallelized progressive network coding with hardware acceleration," in *Quality of Service, 2007 Fifteenth IEEE International Workshop on*. IEEE, 2007, pp. 47–55.

[8] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," in *In ACM Trans. Math. Soft., 5 (1979), pp. 308-323*, 1979.

[9] J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN Basic Linear Algebra Subprograms," in *In ACM Trans. Math. Soft., 14 (1988), pp. 1-17*, 1988.

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney *et al.*, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[11] G. K. and G. R., "On reducing tlb misses in matrix multiplication," The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-2002-55, 2002. [Online]. Available: citeseer.ist.psu.edu/goto02reducing.html

[12] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–27.

[13] J.-G. Dumas, P. Giorgi, and C. Pernet, "Dense linear algebra over word-size prime fields: the fflas and ffpack packages," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 19, 2008.

[14] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, G. Villard *et al.*, "Linbox: A generic library for exact linear algebra," in *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*. World Scientific, 2002, pp. 40–50.

[15] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[16] L. Karlsson, "Computing explicit matrix inverses by recursion," Ph.D. dissertation, MS thesis, Umea University, Department of Computing Science, Sweden, 2006.

[17] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "High performance matrix inversion based on lu factorization for multicore architectures," in *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*. ACM, 2011, pp. 33–42.

[18] H. P. Anvin, "The mathematics of raid-6," 2007.

[19] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel simd instructions." in *FAST*, 2013, pp. 299–306.